AccountVault

Dan Blossom
Marist College
Professor Pablo Rivas
MSCS 630 Security Algorithms and Protocols
May 12th, 2019
Final Writeup

Abstract: This paper is the final writeup for an AES (Advanced Encryption Standard) application I developed to demonstrate its use as a secure way to store confidential information. This information will be encrypted using AES at 128 bits with a randomly generated key (one per ‘account’) and stored in a SQLite Database. This application isn’t intended for ‘cloud’ use or to interact with a users account (example a bank account) but merely a way to store easily forgotten information about that account locally and securely.

Introduction: This project was created to determine if it’s a feasible solution to use AES (Advanced Encryption Standard) in order to securely store personal information. Similar to a password manager application, AccountVault will store information about a users account. However, the main difference is it’s not to interact with the account in anyway (like a password manager). AccountVault is merely a note taking style application where the user enters ‘notes’ about their account. That information is encrypted using AES and stored in an SQLite database. The AES Key used to encrypt the data is then encrypted itself utilizing the Android Keystore feature.

Development: I created the application for modern phones running Android. The development environment used was Android Studio, which is the official IDE for Android Development. Android Studio is actually built off IntelliJ IDE which, in my opinion, is also a very well written

IDE; having IntelliJ at the core is what helps it succeed. Two main programming languages are supported with Android development and Android Studio, Kotlin and Java. The code written for this project was done in Java. I never created an Android application so took this opportunity to understand the development process and compare it to iOS development in Swift. (I'll not be comparing the two here.)

Background: There are plenty of password manager applications, ask a group of 5 people what they use, you're likely to get 5 different answers for 5 different reasons. Those different reasons is why I wanted to develop AccountVault and plan on continuing its development. Example, when asked to provide security questions and answers, I typically provide an answer that's not directly related to the question. I also find that some companies like United Airlines have you use a cryptic username associated with your account number; mileage plus number in United's case. While a password manager would handle that use case, not everyone feels safe using them and rather jot down on a sticky pad and stick it to their monitor. That's where AccountVault comes into play you can put that 'note' into the app and it'll be stored safely and securely. There is no cloud, there is no linking to the account, it's just a digitally secured note about the account.

Methodology: The methodology used to develop AccountVault is that a user will enter information about an account they want to have a 'note' on and it will be encrypted using AES and stored into a SQLite database locally on the Android device. The key used to encrypt the data is randomly generated with the help of Java's SecureRandom class. 32 randomly chosen integers are used to build a 128-bit encryption key to secure the details entered about the account. Each account has its own uniquely created 128-bit random key. After the details are encrypted, they are stored in a local SQLite database. The randomly generated key is then encrypted using Android Keystore and placed in the database for retrieval when required. Using

the Keystore provides an additional layer of security than the initial design proposed. When the user selects an account from the main screen which is created from a list view, another screen will open displaying the stored note. First the 128 bit encryption key used is decrypted using Android Keystore and then the key is used to decrypt the data using a self implemented AES decryption algorithm.

Experiments and Development Process: There were several experiments performed to complete this project. One of the first experiments was to determine if I could create a simple block cypher algorithm utilizing an existing self implemented AES encryption algorithm. The following method previously created allowed me to encrypt 128 bits with an 128 bit key.

```
public String AES(String pTextHex, String keyHex)
```

pTextHex is the plain text in hex and **keyHex** the key in hex.

The goal was to create a way to call that method without changing the underlying details and allowing for any length string of characters not just 32 hex characters.

The solution was to create a wrapper method:

```
public String encrypt(String inputText, String key)
```

inputText is text of any length and **key** is the encryption key.

That method takes the input and breaks it into 16 character chunks which translates to 128 bits. Each character is 8 bits also known as 1 byte which translates to 128 bits. If a chunk is less than 16 characters it's padded. Those 16 character chunks are put into an array, one entry per chunk. After, the AES method is called appending the results to a StringBuilder object which is returned to the caller.

The next stage in the experiment was to decrypt the encoded message. This was done by creating several methods that did the reverse of the encryption method.

The following method decrypted 128 bit chunks of hex data:

`public String aesDecrypt(String encryptedText, String encryptionKey)`

`encryptedText` `is an encrypted hex string and` **`encryptionKey`** `is the key.`

However, creating the decryption function and its helper functions resulted in similar restrictions as the AES(String, String)function where it can only decrypt 128 bit chunks in hex and would return only a hex string and not its character representation. This led to creating a similar function as encrypt(String, String) where a hex string of any length could be passed, it would call aesDecrypt(String, String) under the covers breaking the hex string into 128 bit chunks and decrypting. It would then convert those chunks to its character representation using Java's StringBuilder and returning the decoded test.

`public String decrypt(String outputText, String key)`

`outputText` `the returned decrypted text and` **`key`** `is the key.`

Once those initial functions were in place, I was then able to encrypt and decrypt text of any length.

Here are some initial tests:

```
$ java Driver < test2.txt
      Original: hereisastringlongerthan16charsithink
      Encrypted:
      87E7D255FFCBB81BA269500914CCCA1C33FC048AE140C975FF847F924F
      7A506FF9079715D75E9A1CA28333DF55903CB7
```

```
        Decrypted: hereisastringlongerthan16charsithink
$ java Driver < test3.txt
        Original: yay we can encrypt and decrypt long text!!
        Encrypted:
                53CF80DA1C3712BA4019B8C39F2A53A907B86295BF9347067EB83
                4191376186F3E9282C50F5B87394F1D191A8BC8FFF7
        Decrypted: yay we can encrypt and decrypt long text!!
```

Once the initial tests were complete, the next phase of experimentation was to get a prototype app designed which would preliminarily provide 3 main functions. 1) A main screen that would display a list of accounts the user already added. 2) An add screen, which when the user pressed the add button on the main screen it would present a screen where they can enter new account details. 3) A view account screen which, after a user clicked an account on the main screen would display the details about the account. The final experiment was to integrate the encryption into the app. Once the three main functions were created, encryption and decryption was successfully integrated by calling functions mentioned above.

A Test with encryption enabled but not decryption:

```
myemail@yahoo.com -> 8AC4D6776910172F6D36F01CECE8A97
chicken soup rules. -> 098341F138DA998DB5AFD3DEB6F4AD37EE22D8
```

Once decryption was enabled the text displayed without the user having any knowledge it was once encrypted. The last of my experiments was to solve the problem of storing the AES key with the account information in the database. This was done by utilizing the Android Keystore API and storing the encrypted information in the SQLite database which was decrypted by the

Android Keystore when needed. This allowed me to keep the AES key with the associated account while keeping it a secret.

Discussion: Creating AccountVault to learn about AES encryption and decryption was both challenging and fun. There were several technologies I was unfamiliar with such as: Android Studio, Android Development, AES Encryption, and Java APIs related to security (Android's Keystore, & SecureRandom). While my initial requirements stated I'd be storing the AES key unencrypted in the database I had extra time to either focus on features of AccountVault or solve the key issue and felt the key issue was more of a priority. While the Keystore API isn't too complicated, however, it did take more time than I expected to understand it. Learning Android Development and Android Studio also came with their own set of challenges, such as understanding the Activity Class and how multiple Activities interconnect. While using the AES created for this course is secure since it is nearly impossible to break via brute force it could contain bugs not known to this author. Those bugs could present a security hole if they are exposed. If I decide to take this application further, switching to something like the Android Keystore would be imperative. Android Keystore or similar Android based algorithms are constantly being updated for bugs and adding features. I feel the goals I set out to accomplish were met, to create a functional application to store digital 'notes' about an online account safely and securely using AES and AccountVault does exactly that.

Future Enhancements & Current Limitations: AccountVault has a lot of limitations, for one you're restricted to only storing: username, email address, password, account number and one security question. That needs to be added as a future enhancement to allow the user to add (within reason) any number of items they may want to store. Another limitation is the user cannot edit or delete the notes, just add another. That would be the first feature added, to allow editing and deleting. I would add another layer of security, fingerprint authorization or password

to prevent someone physically holding the device to view the confidential information. Lastly I'd make a lot of style edits allowing it to be more appealing and professional looking.

Conclusion: This was an excellent project that introduced me to AES within Android Development. I overcame a lot of obstacles, one of the most challenging being the ability to decrypt the encrypted text. It can be said loosely it's 'just the reverse' but there is a bit more to it. Implementing the Rijndael Mix Columns and understanding exactly how they fit into the overall algorithm took a lot of research. I look forward to continuing the development of this application and making some of the changes mentioned in the Future Enhancements & Current Limitations section. The first enhancement being to implement Android security and encrypting the database.

References: Similar to the milestone report no scholarly references but I'll include research findings that really pushed this project forward.

https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Rijndael_mix_columns.html

https://crypto.stackexchange.com/questions/2569/how-does-one-implement-the-inverse-of-aes-mixcolumns

https://en.wikipedia.org/wiki/Rijndael_MixColumns

https://developer.android.com/training/articles/keystore

https://developer.android.com/training/basics/firstapp/building-ui

https://developer.android.com/training/basics/firstapp/starting-activity

https://medium.com/@josiassena/using-the-android-keystore-system-to-store-sensitive-information-3a56175a454b